本文欢迎转载，但是请注明出处：<https://github.com/lygyue/Books>

C++基础

本文不打算讲得太深入。如果讲得太深入，请参照C++ Primer。这本旷世字典我基本没看过。

本文也不讨论烂大街的东西，也绝对不讨论语法糖。

但是，我认为，如果连本文都看不明白，理解不透彻，那么搞图形学是不会有前途的，最多也就做一个API程序员，引擎程序员，因此，放弃吧，你不适合这个，还不如写写脚本写写java，照样可以有一个光明的未来（此处为本书第一次劝退，后续估计还有无数多次）。作为程序员三大浪漫之一，需要深入计算机图形学，必然需要跨过一座又一座的高山。这仅仅是第一座！

第一个基础中的基础：二进制。

一个int四个字节

一个float四个字节

一个double八个字节

一个short两个字节

一个char一个字节

任何指针都是四个字节

一个字节8位

字节在英文里是byte，位是bit

是不是觉得超级简单，然后随便一个菜逼都会？那么，是时候来演绎一下了。

问题：一个char的取值范围是多少？一个int的取值范围又是多少？

一部分大大大大菜逼琢磨了半天，实在是想不明白，这个怎么跟取值范围有关了？怎么就知道取值范围了呢？

这里，开始了本书的第二次劝退。放弃吧，图形学不是这样的天赋能搞定的，如果这都能搞定，图形学绝对不是什么三大浪漫了，是时候转行去写脚本了。

我相信，这里难不住很多人，很简单，二进制一位就是开或者关，分别表示0或者1，1位可以取值0或者1，2位是0，1，2，3（二进制是11），那么8位就是2^8-1，最大不就是255吗？这个答案虽然不对，也至少明白了思路。无符号单字节最大是255，由于需要一位描述正负，所以只能是-127，127之间。（实际是-128-127，这里只讨论一个思路，不打算讨论原反补码，理解一下大意即可）

由此类推，那么short，int的取值范围呢？依葫芦画瓢，估计很多人都明白了，然后觉得自己屌炸了，要上天了。

那么再来演绎一下。

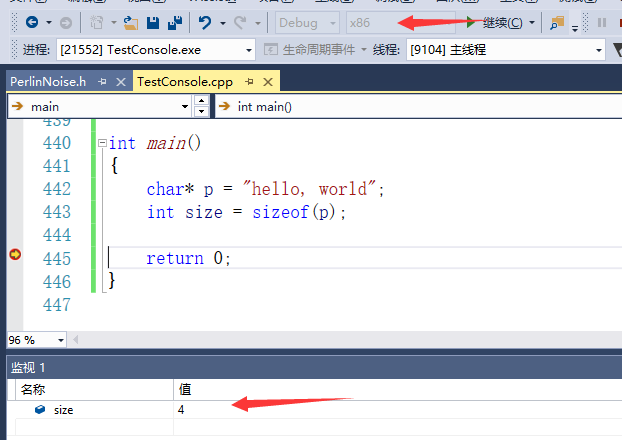
问题：我们常说32位操作系统，64位操作系统，有什么区别？我们为什么需要从32位操作系统升级到64位？

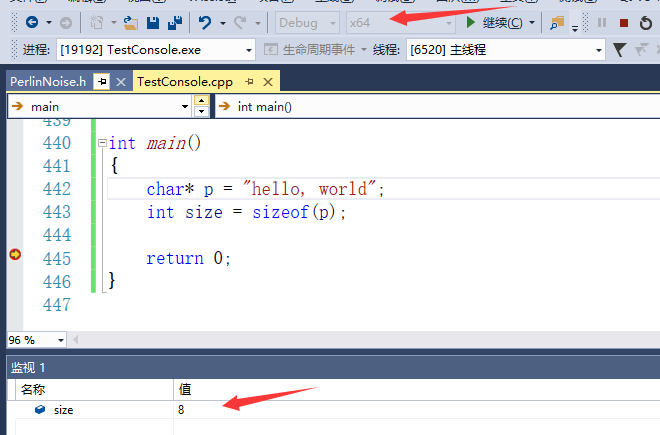
这个问题，我面试的时候有时候会问到。听到过很多回答，例如速度更快（为什么更快？数字大就更牛逼？不知道），例如32位不能满足需求（为什么？不知道）。

本书的第三次劝退，我决定放在这里。不说有没有看过操作系统，稍稍关注一下日常科技，少玩无脑游戏，再结合上面的知识，都比较容易推出来一个结论：

2^32 = 2^10 \* 2^10 \* 2^10 \* 2^2，很简单的指数计算。大概就是4,000,000,000.

这个数字是不是很耳熟？没错，这其实就是4G。32位操作系统，由于操作系统规定了所有指针的大小是32位，那么最大寻址只能是4G。也就是说，32位的操作系统，内存最大必然不能超过4G。超过4G已经无法寻址了。所以以前的xp操作系统，内存绝对不可能大于4G（有64位xp系统，不过很少人用过）。而64位的操作系统，如你所见，一个指针占了8字节，64位。因此，理论内存上限非常大。这里面，限制的就不是位数，也是OS本身的设计了。上两张图佐证一下：





讲到这里，大佬们不屑一顾（多简单，我早就知道了），菜鸟们陷入了深深的沉思。然后豁然开朗，我怎么就没想到呢？

明白了这些关键之后，菜鸟兴奋了，感觉自己打开了新世界的大门，浑身充满了力量，前途一片光明。

这里需要演绎两个问题来泼一下冷水，让菜逼们明白前途多艰。为什么是两个问题而不是一个？因为这两个问题代表了演绎的两个方向。

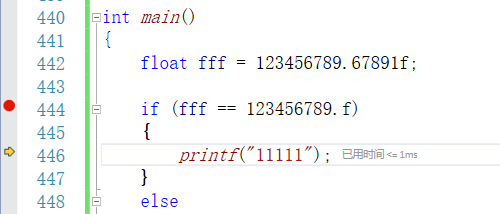
Q1：浮点数取值范围更大还是整数取值范围更大？为什么？

Q2：指针作为一个32/64位的正整数，指向char的指针，指向int的指针，指向结构体的指针，无数的指针，岂不是可以互相随意转换？

第一个问题，菜逼陷入了沉思。但是仔细想一想，都是32位，浮点数还有小数点，岂不是浮点数取值范围更小？对，一定是这样。

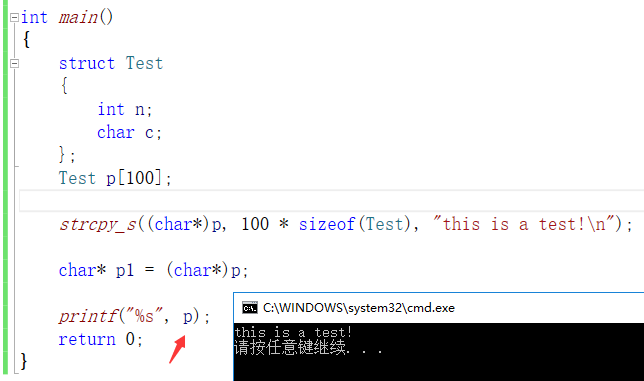
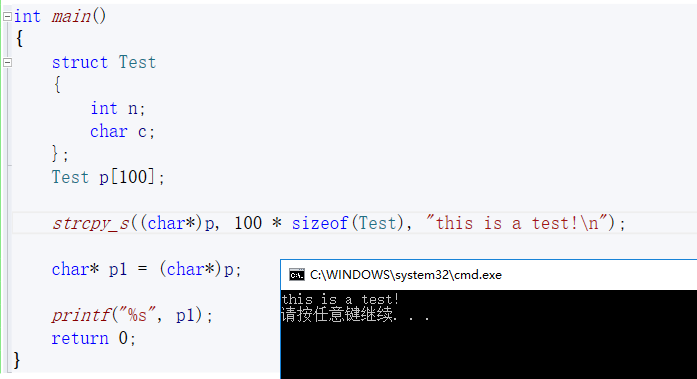
只能说，菜逼太天真了。

浮点数的储存，有一个IEEEE标准，具体细节这里不讲了，讲的话又是去抄书。结论是，浮点数的范围非常非常大，比整数大得多。但是浮点数的精度是个大问题。写这个问题的初衷，是为了让菜逼们深刻记住：浮点数的精度非常非常重要，自始至终贯彻于整个计算机图形学，包括采样精度，包括知名的z fighting问题，也是由浮点数精度造成。

这里先简单演示一下菜逼们容易犯的错误：

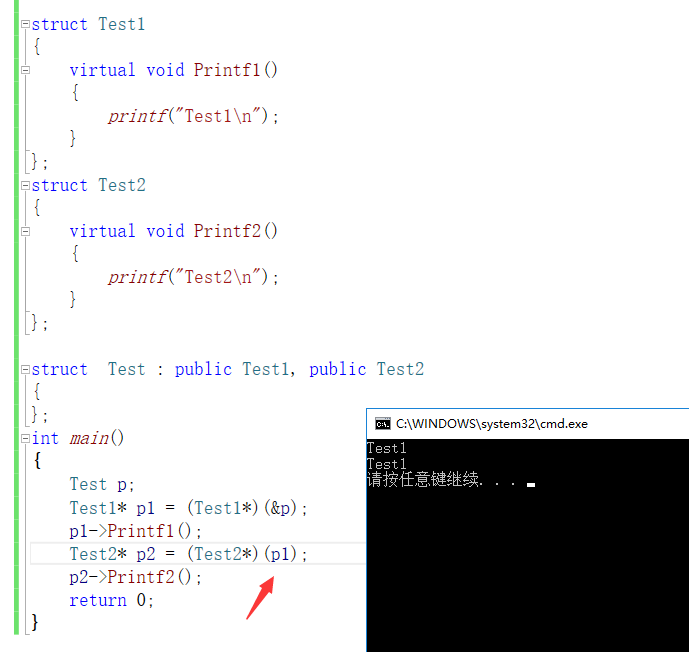
看看这图，越大的数据，精度损失越大。虽然浮点数取值范围可以很大，但是越到后面，可能跟你的预期越远。导致的结果是：两个相差很大的数，可能在内存中居然是一样的；两个浮点数相除，得到的结果居然不是你想要的，诸如此类，不一而足。Z fighting的重面/邻面闪烁，就是这个原因。

第二个问题，菜逼一般不敢下结论，需要做很多的测试，看看这样一个测试：



看到这个测试，心里踏实了。是的，我又懂了，所有指针仅仅是一个正整数，是可以任意转换的，我想怎么用就怎么用！

平心而论，到了这个地步，已经脱离了刚刚入门的菜逼行列。但是，这个理解就是对的吗？还是那句，太天真！



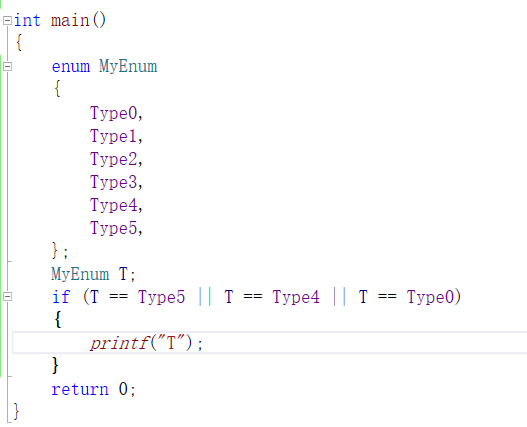
一图胜千言，你以为你已经掌握了真理，真理却给了你一个耳光。

这里，我无意深入去探讨C++的多态机制，只是说，C++的强制转换，在绝大多数的情况下，是没有什么大问题的。但是，还是会有问题的时候，切记慎用。

结论：做人不能太得瑟，就算你写了十几二十年C++，还是低调一些，搞不好什么时候给你当头棒喝。

这一章的最后，用一个常用的用法，去怼群主的吐槽。顺便加深各位菜逼对二进制的理解。

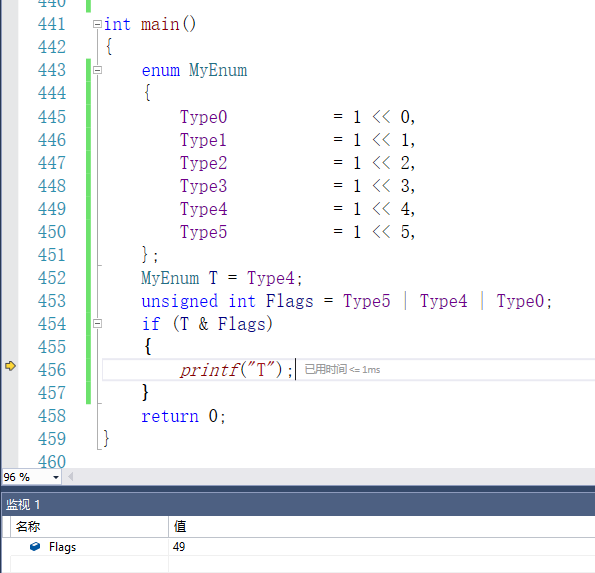
假设一个游戏里面很常见的需求：



这里，最正常的新手写法，显然就是上图这样，做多个判断，群主吐槽的是写了很长一堆判断的，严重影响了可读性。

有什么解决办法吗？群主给出的解决方法，是一个数组，或者是哈希表，直接设定条件到数组或者表，然后查询是否满足条件。

C++里面大量使用的方法大概是这样子的：



这种原理非常简单，就是个简单的二进制位运算。长篇累牍的介绍原理，实在是有点索然无味，看不懂的菜鸟可以去百度一下二进制运算吧。

这类东西本质上跟图形学毫无关联。不过，在微软的D3D代码里面，有大量差不多的运用，如果这种没搞懂，后续看DX的接口也会事倍功半。

C++基础之二，内存

早期写Basic代码的时候，完全没有内存的概念。变量甚至都不考虑类型，想怎么用就怎么用，想怎么写就怎么写，简单暴力。

然而，C++而言，内存是重中之重。一旦这块理解不够深刻，各种乱七八糟的问题就来了，内存泄漏，内存越界……。

就我们编写C++程序而言，内存一般分为三种：

静态存储区：内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据、全局数据和常量。

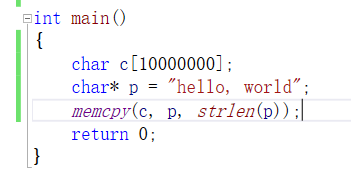
栈区：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆区：亦称动态内存分配。程序在运行的时候用malloc或new申请任意大小的内存，程序员自己负责在适当的时候用free或delete释放内存。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。 但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉，否则，我们认为发生了内存泄漏现象。

以上解释来自百度，懒得自己写一遍了。

看了这样一段文字，是不是觉得自己已经理解到了内存的真谛，醍醐灌顶？

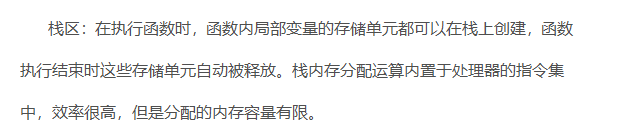
那么，来演绎一下，请看下图：



如此简单的代码，请问，有错误吗？

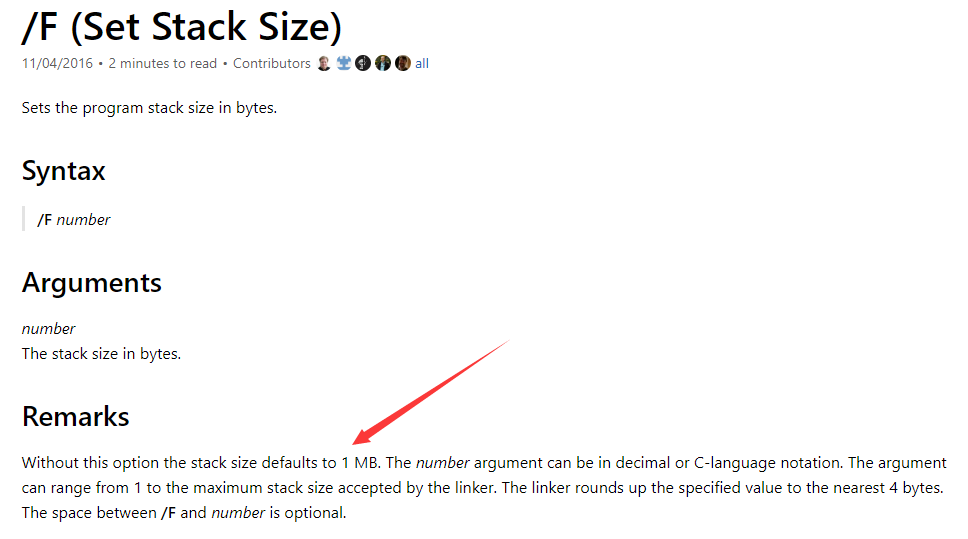
菜逼有点心慌，不急，先打开VS，输入，编译，哇靠，0错误0警告，妥妥的正确代码，没毛病啊,，连编译器都站我这边，告诉我这没有错。

然而，程序刚启动，崩溃了。心里一万匹草泥马奔过。人生苦短，还是转python算了，保证没有各种乱七八糟看不懂的问题。

是的，这么写就是错误的，我们再来复述一下这段话：

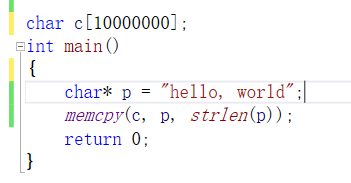
首先，我们确定，任何一个变量都是要占用内存的。所有变量/数组/指针/容器……全是要占用内存的。那么，栈内存有多大？这里只说有限，到底有多有限？各类blog，各类文章基本上语焉不详。所谓天下文章一大抄，一大堆写blog的，写文章的，都是边学边写，顺便做笔记，导致了一大堆文章良莠不齐。（此处应加上一篇文章《我为什么说各种blog文章一堆渣渣》）。

显然，我们不能做这样一个渣渣，我们有图有真相。我基本只做windows开发，那么我只找windows的真相，其他真相你们自己找。



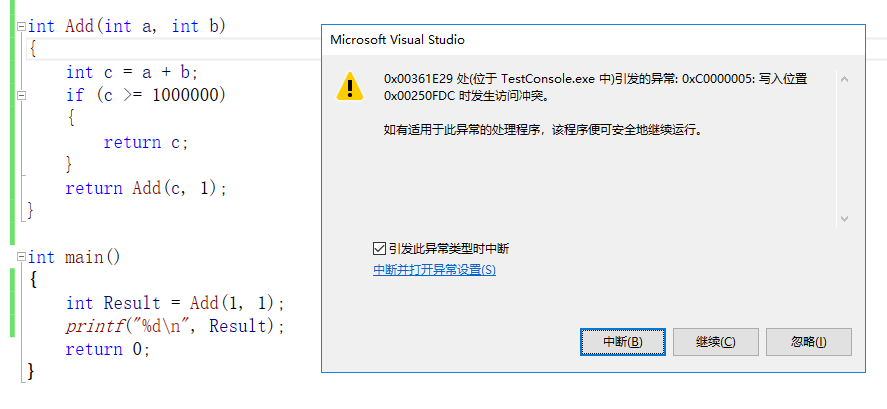
在微软的官网，这里被安排得明明白白。无论是编译器的设计，还是Thread Stack Size，都明明白白的写着1M。因此，一个稍大点的数组，都不该直接在函数里面定义。

不在函数里面这样用，那么应该怎样用呢？这样可以吗？



这样写，编译，运行，都没有错。但是，这样写的话，这块内存会一直被占用，而除非程序结束，才会被释放。如果你确定你是这样的需求，那么无所谓了。一旦不是，我们就该用堆内存。

栈内存的使用，还有一个误区，那就是递归。且看这个代码：



毫无疑问，这看似正确的代码，执行的时候，崩溃了。原因很简单，栈内存耗尽。栈内存的意思是，任何一个函数开始调用的时候，就会开始使用栈内存，除非这个函数执行结束，栈内存并不会被回收。而递归调用使用的栈内存，其实是所有函数使用的栈内存相加。

这里，Add函数三个变量，一个4字节，一个递归使用了12字节内存，总共栈内存才1M，不需要太多次的递归，就会造成栈内存耗尽而崩溃。

因此，程序之中尽量少点使用递归，尤其是那些无法确定递归次数的算法，典型的例如深度搜索算法。我曾经在我以为我比较牛逼的时候，写一个深度搜索算法来计算图片的轮廓，然后导致了栈内存耗尽，印象深刻。

总结一下：内存主要有三个来源，一个是编译的时候就决定了的，自始至终被应用程序占用的静态内存；一个是栈内存，比较小，函数体内部直接使用，英文是stack；另外一个是需要用到的时候再申请，不用了就释放的内存，英文是heap。那么，这样设计有什么好处吗？

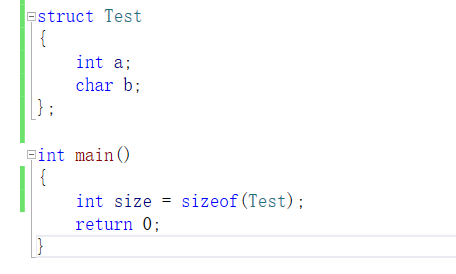
这是一个非常常见的设计，贯彻于我们生活的始终。举例：

你去读大学，学费+住宿费是10000，你父母每个月给你的生活费是1000，当你有其他大笔开销，例如买电脑，买手机……再问父母要。

这就是一个一样的设计。开局，你必须花那么多，那么先给你。然后每个月再给你一部分，这属于日常开销。你需要其他额外开销，再跟父母申请。父母穷，不给你，就好比new一块内存失败了（电脑配置太差）。

这样的方式，最大限度的利用了资源，提升了效率。有人会说，new一块内存还会失败？不是说内存不够了会用硬盘作为虚拟内存吗？当然会，不信你在xp底下new一块2G的内存试试看？

内存对齐。



这里，结构体占了几个字节的内存？不知道内存对齐的，毫不犹豫认为是4 + 1 = 5；

事实上，VS默认字节对齐的情况下，结果是8.

字节对齐有什么好处？

这里只说结论，不解释了，网上烂大街。

第一个好处：速度更快。这里的速度，主要是指cpu对数据的处理速度。

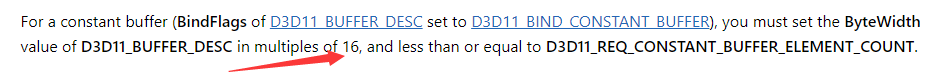
第二个好处：跨平台或者不同进程之间的数据通讯/网络通信等等，不使用对齐的话，可能会导致对数据的存取出错。

老实说，我在写这个对齐的时候，都是一边吐槽一边写的，因为这看似一个烂大街的，人人知道的东西，我为什么还要单独写一写？这个都不知道的，图形学早就该劝退了对不对？

这主要还是要考虑到一些不知道这个的菜鸟，对图形学还不死心。在DX12里面，或者说在后期的GPU编程里面，非常多的地方都涉及到这个对齐，如果这些都不知道，后续会碰到非常多的问题。例如：

1、BMP图片，最简单的位图，就有一个四字节对齐的问题。这个不知道的，我会在图像基础里面详细讲一下，或者自己去百度。

2、DX里面非常常见的const buffer，创建的时候，必须16字节对齐，不对齐的直接创建失败。详情请看这里：<https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11device-createbuffer>



1. 做SIMD优化的时候，所有的结构体，必须16字节对齐。这一块，是一般CPU程序优化的常见套路，并且略高大上。想深入理解的可以自行百度，我会在后续的优化篇章里面详细讲解一下。

一句话总结一下：那么多大公司，那么多大佬，不约而同全部要求你要搞对齐，那么，你就不要一个人对抗整个世界了，那样孤胆英雄的造型虽然看起来很酷，往往死得快。

内存的速度比较

这里往往涉及到一个问题：堆内存快一些，还是栈内存快一些，还是静态 储存区更快一些？

被问到这个问题的人，往往一脸懵逼，都是内存，速度还不一样吗？

而我在面试中问到这个问题的时候，往往都有了被打的觉悟，实在是太偏门了，一般人完全不需要知道，或者是完全用不到！

但是，要成为一个大神，就是要知道一些别人不知道的，并且很有用很关键的东西。这个东西在计算机图形学里面，完全用不到。但是，当你深入理解了这些理论，往往比较容易理解显卡的设计意图，更容易理解一些硬件相关的东西，在后面我解释显卡的一些参数，设计的时候，会有相关的论述，这里，暂且就事论事，讲讲内存的速度问题。

毫无疑问，绝大多数情况下，栈内存是最快的。原因？很简单，内存，仅仅是储存器的一种，就物理储存器而言，一台PC上有N多可以储存数据的硬件，从CPU访问速度而言，从高到低，包括但不限于：CPU寄存器，CPU一级缓存，CPU二级缓存，CPU三级缓存，内存，硬盘缓存，硬盘。

关于这些东西的详细描述，随便百度即可。栈内存作为最经常使用的内存块，往往很容易就会进入CPU缓存。也就是说，这块的访问，很可能物理上是在CPU缓存的。而堆内存相对较大，并且访问频率没那么高，大多数时候，是不如栈内存效率更高的。

关于这块的程序优化，我听过一个最丧心病狂的优化方式，是迅雷某产品老大跟我讲过，他们会对某一块硬盘不断读，让这块硬盘进入硬盘缓存，从而提升加载速度。我深深的鄙视了他，然后觉得这个方法说不定还能用一下：（。

这块最常见的体验：打开某大型程序，等半天，打开了。然后关掉，再打开，尼玛，速度真快。大概就这意思。

好了，关于C++基础部分，暂时只讲到这里。后续想到其他跟图形学相关的东东再补充。至于像什么设计模式，各种语法糖，跟图形学毫无关系的，自己找资料看即可。算法类的跟图形学有点关系的，例如四叉树，八叉树，感觉讲了也没什么意思，这类已经脱离了C++的范畴。